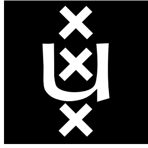


University of Amsterdam

Programming Research Group

Tuplix Calculus Specifications of Financial Transfer Networks

---

J.A. Bergstra
S. Nolst Trenite
M.B. van der Zwaag

Report PRG0805 May 2008

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7591
e-mail: janb@science.uva.nl

S. Nolst Trenite

Faculty of Science
University of Amsterdam

e-mail: sanne@science.uva.nl

M.B. van der Zwaag

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7584
e-mail: mbz@science.uva.nl

Programming Research Group Electronic Report Series

Tuplix Calculus Specifications of Financial Transfer Networks

Jan A. Bergstra[1*] Sanne Nolst Trenité[2]
Mark B. van der Zwaag[1]

[1]Section Software Engineering, Informatics Institute, University of Amsterdam

[2]Faculty of Science, University of Amsterdam

Email: {janb,sanne,mbz}@science.uva.nl

Contents

1 Introduction **1**

2 Financial Transfer Networks **2**

3 Flux over a Network **4**

4 Visualizing Internal Streams **7**

5 Function Definition and Binding **9**

A Derivations **10**

B Primer on Tuplix Calculus **13**

B.1 Cancellation Meadows 14

B.2 Basic Tuplix Calculus 15

B.3 Zero-Test Logic 17

B.4 Generalized Alternative Composition and Auxiliary Operators . . 18

1 Introduction

In [3] we described the application of Tuplix Calculus (TC, see [6]) in the formalization of financial budgets. Here, we explore this application further starting with the definition of financial transfer networks. We consider the notion of flux of money over a network, and define a flux constraint operator that enforces matching influx and outflux for units. We exploit so-called signed attribute

*Partially supported by the Dutch NWO Jaquard Project Symbiosis, project number 638.003.611.

notation to make internal streams visible through encapsulations. Finally, we propose a Tuplix Calculus construct for the definition of data functions. We assume familiarity with Tuplix Calculus; its syntax and axioms are collected in Appendix B.

2 Financial Transfer Networks

Implicit starting point in the modular budget design in [3] is the assumption of an underlying (organizational) structure: tuplix expressions specify budgets for certain parties, and by composition we obtain budgets for larger parts (of an organization). Of importance is also the identification of attributes, that are used in the specification of payments between parts, or between parts and external parties.

Example 1. As a simple example, consider an organization consisting of parts P and Q, and assume that attribute a is used to specify payments between these parts. Using the names P and Q also as tuplix meta-variables, we define

$$P = a(10), \quad Q = a(-10).$$

So, P will pay amount 10, while Q intends to receive amount 10. When we compose P and Q, expressed as $\partial_{\{a\}}(P \oplus Q)$, these entries synchronize successfully.

We find it worthwhile to introduce a mathematical format for organizational structures. We define a *financial transfer network* (FTN) as a set of *units* with in-going and outgoing channels: a channel is a directed link between units, or between a unit and an external party, that is labeled with an attribute. Labels of in-going channels of a unit are used in the specification of payments to the unit, and the labels of outgoing channels are used to specify payments made by the unit. We require that any channel is in-going for at most one unit and outgoing for at most one unit.

Definition 1. An FTN consists of:

1. a set Attr of attributes;
2. a set Unit of units;
3. a function $\mathit{in}\colon \mathit{Unit} \to 2^{\mathit{Attr}}$;
4. a function $\mathit{out}\colon \mathit{Unit} \to 2^{\mathit{Attr}}$;

such that for all distinct $g, h \in \mathit{Unit}$, $\mathit{in}(g) \cap \mathit{in}(h) = \emptyset$ and $\mathit{out}(g) \cap \mathit{out}(h) = \emptyset$.

An attribute a is *internal* if there are units $g, h \in \mathit{Unit}$ with $a \in \mathit{in}(g) \cap \mathit{out}(h)$. An attribute is *external* if it is not internal.

An FTN can be depicted in a graph-like manner, with units as nodes, and arrows (called channels) labeled with attributes between units, or between a unit and an external party. Because an attribute of an FTN can be the label

of at most one channel, we shall also speak of the channel a, rather than the channel labeled with attribute a. A channel is internal if its label is internal; this is the case if it connects units of the network, see the following example.

Example 2. Consider the FTN with $Attr = \{a, b, c\}$, $Unit = \{g, h\}$, and

$$in(g) = \{a\}, \quad out(g) = in(h) = \{b\}, \quad in(h) = \{c\}.$$

This network is depicted as

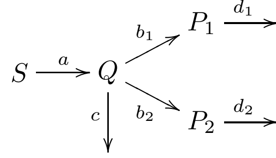
$$\xrightarrow{a} g \xrightarrow{b} h \xrightarrow{c}$$

The channels a, c are external, b is internal.

Given an FTN, a *specification* of a unit g is a tuplix expression P_g that uses only the elements of $in(g) \cup out(g)$ as attributes.

Example 3. This example is a shortened, simplified version of the example presented in [3]. We have added the presentation of the organizational structure as an FTN.

We consider an FTN as depicted in the following picture:



The units and their specifications (for a given period of time, e.g., the calendar year 2008):

- S is a *financial source* that rewards production: for each product that is produced, a constant reward *rew* is allocated to unit Q. For production unit P_i (see below) the data variable n_i stands for the number of products produced by P_i during the period that is covered.

  Specification:

  $$S \stackrel{\text{def}}{=} a(rew \cdot (n_1 + n_2)).$$

- The *control* unit Q will dispatch the rewards to the production units after deduction of a fixed fraction k (a value between 0 and 1) that is paid via c to an external service center. It further distributes the remainder of the rewards equally among the production units:

  $$\begin{aligned} Q \stackrel{\text{def}}{=} \textstyle\sum_x (&a(-x) \between \\ &c(k \cdot x) \between \\ &(1-k) \cdot (b_1(x/2) \between b_2(x/2))). \end{aligned}$$

- The *production* units P_i, for $i = 1, 2$, receive money from Q via b_i and pay for their expenses via d_i (in this simplified example, these units act as serial buffers only, that is, they simply pass on what they receive):

$$P_i \stackrel{\text{def}}{=} \sum_x (b_i(-x) \oplus d_i(x)).$$

A combined budget B is specified by the encapsulated composition of these specifications:

$$B \stackrel{\text{def}}{=} \partial_{\{a,b_1,b_2\}}(S \oplus Q \oplus P_1 \oplus P_2).$$

The encapsulation enforces synchronization on the internal channels and then hides these internal streams (in Section 3 we elaborate on the notion of streams).

We find (see Appendix A for the derivation):

$$\begin{aligned} B = \textstyle\sum_x (&\gamma(x = \mathit{rew} \cdot (n_1 + n_2)) \oplus \\ &c(k \cdot x) \oplus \\ &(1-k) \cdot (d_1(x/2) \oplus d_2(x/2))). \end{aligned}$$

Alternatively, we may redefine Q so that it pays the production units proportionally to their contribution to the total production:

$$\begin{aligned} Q \stackrel{\text{def}}{=} \textstyle\sum_x (&a(-x) \oplus \\ &c(k \cdot x) \oplus \\ &(1-k) \cdot x \cdot (b_1(n_1/(n_1+n_2)) \oplus b_2(n_2/(n_1+n_2)))). \end{aligned}$$

Then we find, for the combined budget:

$$B = c(k \cdot \mathit{rew} \cdot (n_1 + n_2)) \oplus (1-k) \cdot (d_1(\mathit{rew} \cdot n_1) \oplus d_2(\mathit{rew} \cdot n_2))$$

with a similar derivation.

3 Flux over a Network

Unit specifications of an FTN can be thought of as determining an unrealized flux over the internal channels of a network. Take for instance the channel

$$g \xrightarrow{a} h.$$

We speak of a *stream* over a, when the total amounts specified for a by g and by h match (that is, add up to zero). We then also say that g has outflux over a and h has influx over a. When there is no match, there is no flux; the flux is realized when we compose unit specifications, and encapsulation over the internal attributes is successful.

A very simple example: consider

$$g \xrightarrow{a} h$$

with specifications $P_g = a(t)$ and $P_h = a(-s)$. We say that g has outflux of size t along a, and that h has influx of size s along a. If the outflux of g along a matches the influx of h along a, that is, if t equals s, then there is a stream of this size from g to h. This matching corresponds to the success of encapsulation of the composed unit specifications: we find

$$\partial_{\{a\}}(P_g \oplus P_h) = \gamma(t = s).$$

This encapsulation reduces to an equality test; unsuccessful encapsulation yields the null tuplix δ. Note that encapsulation hides the internal transactions; in Section 4 we look at a way to make successful internal transactions (i.e., *flux*) of units visible.

Flux dynamics comes into play with generalized alternative composition (summation) over amounts. For example, redefine P_h so that it will receive *any* amount, and send this along:

$$P_g = a(t), \quad P_h = \textstyle\sum_x a(-x) \oplus b(x),$$

then we find that successful encapsulation determines the outflux of h:

$$\partial_{\{a\}}(P_g \oplus P_h) = b(t).$$

Working with this perspective we find it natural to be able to require for certain units that 'what goes in also comes out.' For example, specify that h will receive any amount along a and will transfer any amount along b:

$$P_g = a(t), \quad P_h = \textstyle\sum_x a(-x) \oplus \sum_y b(y).$$

Encapsulation over a will enforce the transfer of amount t along a, and an additional requirement that the total flux of h equals zero would turn h into a serial buffer that forwards amount t along b.

We define a unary *flux constraint operator* that does exactly this: it adds to its argument the constraint that its total flux equals zero. This operator (written K, after Kirchhoff) is defined as follows:

$$K(X) = K_0(X) \tag{1}$$
$$K_t(\delta) = \delta \tag{2}$$
$$K_t(\varepsilon) = \gamma(t) \tag{3}$$
$$K_t(\gamma(x) \oplus X) = \gamma(x) \oplus K_t(X) \tag{4}$$
$$K_t(a(x) \oplus X) = a(x) \oplus K_{t+x}(X) \tag{5}$$
$$K_t(X + Y) = K_t(X) + K_t(Y) \tag{6}$$
$$K_t(\textstyle\sum_x P) = \sum_x(K_t(P)) \qquad x \notin FV(t) \tag{7}$$

Example 4. We define periodic specifications for a unit Q and a reserve R. The unit Q receives income from and has expenditures to external parties. Every period it withdraws a fixed amount from R, and it reserves a fixed percentage

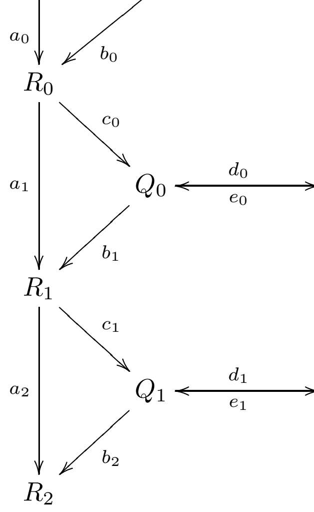


Figure 1: Reserve buffers example

of its income to the reserves of the next period. Any reserves that are not withdrawn are transferred to the next period. The flux constraint operator is used to enforce this transfer of reserves. It is also applied to Q so that it will spend any income that is not reserved.

We make this more precise. We define Q_n and R_n for the unit Q and the reserve R in period n. The following attributes are used:

- a_{n+1} for the transfer from R_n to R_{n+1}
- b_{n+1} for the reservation from Q_n to R_{n+1}
- c_n for the withdrawal from R_n by Q_n
- d_n for the external income of Q_n
- e_n for the external expenditures of Q_n

The network is depicted in Figure 1.

Define

$$R_n = K(\sum_{u,v,w,x} a_n(-u) \oplus b_n(-v) \oplus c_n(w) \oplus a_{n+1}(x))$$

which can be rewritten to

$$R_n = \sum_{u,v,w,x} \gamma(u+v = w+x) \oplus a_n(-u) \oplus b_n(-v) \oplus c_n(w) \oplus a_{n+1}(x).$$

In the specification of Q_n we use the free data variables pw (periodic withdrawal), inc_n (income in period n), and k (reserve fraction, a value between 0

and 1). Define

$$\begin{aligned}Q_n &= K(\textstyle\sum_u c_n(-pw) \oplus d_n(-inc_n) \oplus b_{n+1}(k \cdot inc_n) \oplus e_n(u))\\ &= \textstyle\sum_u \gamma(u = pw + (1-k)\cdot inc_n) \oplus\\ &\qquad c_n(-pw) \oplus d_n(-inc_n) \oplus b_{n+1}(k \cdot inc_n) \oplus e_n(u)\\ &= c_n(-pw) \oplus d_n(-inc_n) \oplus b_{n+1}(k \cdot inc_n) \oplus e_n(pw + (1-k)\cdot inc_n)\end{aligned}$$

Define

$$P_n = \partial_{H_n}(Q_0 \oplus \cdots \oplus Q_n \oplus R_0 \oplus \cdots \oplus R_{n+1})$$

where

$$H_n = \{a_{i+1}, b_{i+1}, c_i \mid 0 \leq i \leq n\}.$$

For P_0 and P_1 we find (see derivations in Section A):

$$\begin{aligned}P_0 = K(&\textstyle\sum_{u,v,w,x}\\ &a_0(-u) \oplus b_0(-v) \oplus\\ &d_0(-inc_0) \oplus e_0(pw + (1-k)\cdot inc_0)) \oplus\\ &c_1(w) \oplus a_2(x)),\end{aligned}$$

$$\begin{aligned}P_1 = K(&\textstyle\sum_{u,v,w,x}\\ &a_0(-u) \oplus b_0(-v) \oplus\\ &d_0(-inc_0) \oplus e_0(pw + (1-k)\cdot inc_0) \oplus\\ &d_1(-inc_1) \oplus e_1(pw + (1-k)\cdot inc_1)\\ &c_2(w) \oplus a_3(x)),\end{aligned}$$

and this generalizes to

$$\begin{aligned}P_n = K(&\textstyle\sum_{u,v,w,x}\\ &a_0(-u) \oplus b_0(-v) \oplus\\ &c_{n+1}(w) \oplus a_{n+2}(x)) \oplus\\ &\textstyle\bigoplus_{i=0,\ldots,n} d_i(-inc_i) \oplus e_i(pw + (1-k)\cdot inc_i)).\end{aligned}$$

4 Visualizing Internal Streams

In an FTN with unit specifications we speak of an internal stream over a channel, if encapsulation over that channel is successful (does not yield the null tuplix δ). In an encapsulation

$$P = \partial_H(P_0 \oplus \cdots \oplus P_k)$$

of unit specifications P_i, all information on internal streams is lost, that is, due to the encapsulation no entries with attributes from H occur in P. Still, it may be useful to see the internal streams of a unit under influence of composition and encapsulation. We shall exploit *signed attribute notation* to retain focus on encapsulated specifications: we add copies of internal entries that will remain visible after encapsulation.

Signed Attribute Notation

So far we have used *flat attribute notation* for entries: for a unit g, if $a \in in(g)$, then an entry $a(t)$ is interpreted as influx of amount $-t$ to g, and if $a \in out(g)$, then $a(t)$ is interpreted as outflux of amount t from g. The notation is neutral in this respect (and this is the basis for the definition of encapsulation).

An alternative is *signed attribute notation*: for attribute a, assume fresh attributes $-a, +a$, and write $-a(t)$ for influx of amount t, and $+a(t)$ for outflux of amount t. We have not defined encapsulation for this notation.

Clearly, tuplix expressions in signed attribute notation can be transformed to flat attribute notation by replacing entries $+a(t)$ by $a(t)$, and $-a(t)$ by $a(-t)$. Vice versa, for a given unit g, transform $a(t)$ to $-a(-t)$ if $a \in in(g)$, and to $+a(t)$ if $a \in out(g)$.

Combined Flat and Signed Attribute Notation

For a unit g and a set of (internal) attributes H, the mapping $\zeta_{g,H}$ will add a signed copy of internal entries of g in a specification using flat attribute notation.

$$\zeta_{g,H}(\delta) = \delta \tag{8}$$

$$\zeta_{g,H}(\varepsilon) = \varepsilon \tag{9}$$

$$\zeta_{g,H}(\gamma(x)) = \gamma(x) \tag{10}$$

$$\zeta_{g,H}(a(x)) = \begin{cases} +a(x) \between a(x) & \text{if } a \in out(g) \cap H \\ -a(-x) \between a(x) & \text{if } a \in in(g) \cap H \\ a(x) & \text{otherwise} \end{cases} \tag{11}$$

$$\zeta_{g,H}(X \between Y) = \zeta_{g,H}(X) \between \zeta_{g,H}(Y) \tag{12}$$

$$\zeta_{g,H}(X + Y) = \zeta_{g,H}(X) + \zeta_{g,H}(Y) \tag{13}$$

$$\zeta_{g,H}(\textstyle\sum_x X) = \textstyle\sum_x \zeta_{g,H}(X) \tag{14}$$

The resulting specification combines flat and signed attribute notation.

Encapsulation

Assume we have units $g_0, \ldots, g_k$ with corresponding specifications $P_0, \ldots, P_k$, and we want to see what composition and encapsulation with $P_1, \ldots, P_k$ do to P_0. Let H be the set of attributes that are internal to $g_0, \ldots, g_k$. The encapsulation

$$P = \partial_H(\zeta_{g_0,H}(P_0) \between P_1 \between \cdots \between P_k),$$

will, if successful, contain signed copies of the internal transactions of g_0. We can now focus on g_0 by letting

$$J = \{a, +a, -a \mid a \in in(g_0) \cup out(g_0)\},$$

and selecting (see definition on page 18) on the attributes in this set:

$$Select_J(P)$$

shows all the transactions of g_0 under influence of the encapsulation.

Of course, we can also make all internal streams of the composition visible:

$$\partial_H(\zeta_{g_0,H}(P_0) \oplus \zeta_{g_1,H}(P_1) \oplus \cdots \oplus \zeta_{g_k,H}(P_k)).$$

Example 5. Consider the following network:

$$\xrightarrow{a} g \xrightarrow{b} h \xrightarrow{c}$$

Take unit specifications

$$P_g = a(-1) \oplus b(1),$$
$$P_h = b(-1) \oplus c(1),$$

and observe that

$$\partial_{\{b\}}(P_g \oplus P_h) = a(-1) \oplus c(1).$$

The encapsulation enforces synchronization on b, and leaves no trace of this synchronization.

Now consider

$$P = \partial_{\{b\}}(\zeta_{g,\{b\}}(P_g) \oplus P_h) = a(-1) \oplus +b(1) \oplus c(1)$$

where the signed copy of the internal outflux of g on b remains visible. Finally, let

$$J = \{a, +a, -a \mid a \in \mathit{in}(g) \cup \mathit{out}(g)\},$$

and find

$$\mathit{Select}_J(P) = a(-1) \oplus +b(1).$$

5 Function Definition and Binding

We extend Tuplix Calculus with a construct to define data functions, and with summation over functions. We only sketch how this extension can be achieved; a fully worked-out technical account is future work. We extend the signature of the data type with lambda abstraction and application in order to express functions. For example,

$$\lambda x.x + x$$

is the function that doubles its argument, and

$$(\lambda x.x + x)2$$

is the function applied to argument 2. Adopting β-conversion as usual, this reduces to $2 + 2$. We also assume standard α-conversion (renaming of bound variables). We further assume for each arity a set of function variables. If f is a function variable of arity k, we write

$$f(t_1, \ldots, t_k)$$

for the application of f to arguments $t_1, \ldots, t_k$. We write $\lambda\bar{x}.t(\bar{x})$ for the lambda abstraction over some given, implicit number of variables x, and $f(\bar{x})$ for the application of f to arguments $\bar{x}$, where the number of arguments is always assumed to be equal to the arity of f.

A function definition

$$f = \lambda\bar{x}.t(\bar{x}),$$

where f is a function variable, is expressed in the Tuplix Calculus by the construct

$$\Gamma(f, \lambda\bar{x}.t(\bar{x})),$$

and we would have, e.g.,

$$\Gamma(f, \lambda x.x + x) \oplus a(f(1)) = \Gamma(f, \lambda x.x + x) \oplus a(2).$$

To derive such identities we adopt the axiom scheme

$$\Gamma(f, \lambda\bar{x}.t(\bar{x})) = \Gamma(f, \lambda\bar{x}.t(\bar{x})) \oplus \gamma(f(\bar{s}) - t(\bar{s})), \tag{FD}$$

for any data terms $\bar{s}$.

Final step: we extend Tuplix Calculus with summation $\sum_f$ over function variables f. This is very similar to summation over data variables.

With these features we can define and use functions in a 'let-like' manner in specifications. The general form

$$\textstyle\sum_f (\Gamma(f, \lambda\bar{x}.t(\bar{x})) \oplus P)$$

may be read as 'let f be defined as $\lambda\bar{x}.t(\bar{x})$ in tuplix P.'

For an example application we refer to [4]. In that paper we define a budget allocation to faculties at a university-level. The allocation for a faculty F can be given by a faculty-independent function f, which takes as input a number of parameter values specific to F. So, say that

$$\Gamma(f, \lambda\bar{x}.t(\bar{x}))$$

defines f, and that the allocation to F is defined as $f(\bar{x}_F)$. The total of budget allocations is then specified by

$$\textstyle\sum_f (\Gamma(f, \lambda\bar{x}.t(\bar{x})) \oplus \bigoplus_F (a_F(f(\bar{x}_F)))),$$

where a_F is a channel name used in the transfer of money to F.

A Derivations

Note: a zero test $\gamma(t - s)$ may be written as $\gamma(t = s)$.

Derivation for Example 3:

$$
\begin{aligned}
B &= \partial_{\{a,b_1,b_2\}}(S \oplus Q \oplus P_1 \oplus P_2) \\
&= \partial_{\{a,b_1,b_2\}}(\\
&\quad a(rew \cdot (n_1+n_2)) \oplus \\
&\quad \textstyle\sum_u (a(-u) \oplus c(k \cdot u) \oplus (1-k) \cdot (b_1(u/2) \oplus b_2(u/2))) \oplus \\
&\quad \textstyle\sum_u (b_1(-u) \oplus d_1(u)) \oplus \\
&\quad \textstyle\sum_u (b_2(-u) \oplus d_2(u))) \\
&= \textstyle\sum_{u,v,w} \partial_{\{a,b_1,b_2\}}(\\
&\quad a(rew \cdot (n_1+n_2)) \oplus \\
&\quad a(-u) \oplus c(k \cdot u) \oplus (1-k) \cdot (b_1(u/2) \oplus b_2(u/2)) \oplus \\
&\quad b_1(-v) \oplus d_1(v) \oplus \\
&\quad b_2(-w) \oplus d_2(w)) \\
&= \textstyle\sum_{u,v,w} (\gamma(u = rew \cdot (n_1+n_2)) \oplus \\
&\quad \gamma(v = (1-k)u/2) \oplus \\
&\quad \gamma(w = (1-k)u/2) \oplus \\
&\quad c(k \cdot u) \oplus d_1(v) \oplus d_2(w)) \\
&= \textstyle\sum_u (\gamma(u = rew \cdot (n_1+n_2)) \oplus \\
&\quad c(k \cdot u) \oplus \\
&\quad (1-k) \cdot (d_1(u/2) \oplus d_2(u/2)))
\end{aligned}
$$

Derivation for Example 4:

$$
\begin{aligned}
P_0 &= \partial_{\{a_1,b_1,c_0\}}(Q_0 \oplus R_0 \oplus R_1) \\
&= \partial_{\{a_1,b_1,c_0\}}(\\
&\quad c_0(-pw) \oplus d_0(-inc_0) \oplus b_1(k \cdot inc_0) \oplus e_0(pw + (1-k) \cdot inc_0) \oplus \\
&\quad \textstyle\sum_{u,v,w,x} \gamma(u+v = w+x) \oplus \\
&\qquad a_0(-u) \oplus b_0(-v) \oplus c_0(w) \oplus a_1(x) \oplus \\
&\quad \textstyle\sum_{u',v',w',x'} \gamma(u'+v' = w'+x') \oplus \\
&\qquad a_1(-u') \oplus b_1(-v') \oplus c_1(w') \oplus a_2(x')) \\
&= \textstyle\sum_{u,u',v,v',w,w',x,x'} \\
&\quad \gamma(u+v = w+x) \oplus \gamma(u'+v' = w'+x') \oplus \\
&\quad d_0(-inc_0) \oplus e_0(pw + (1-k) \cdot inc_0) \oplus \\
&\quad a_0(-u) \oplus b_0(-v) \oplus c_1(w') \oplus a_2(x') \oplus \\
&\quad \partial_H(c_0(-pw) \oplus b_1(k \cdot inc_0) \oplus \\
&\qquad c_0(w) \oplus a_1(x) \oplus a_1(-u') \oplus b_1(-v'))
\end{aligned}
$$

$$
\begin{aligned}
&= \textstyle\sum_{u,u',v,v',w,w',x,x'} \\
&\quad \gamma(u+v=w+x) \oplus \gamma(u'+v'=w'+x') \oplus \\
&\quad d_0(-inc_0) \oplus e_0(pw+(1-k)\cdot inc_0) \oplus \\
&\quad a_0(-u) \oplus b_0(-v) \oplus c_1(w') \oplus a_2(x') \oplus \\
&\quad \gamma(w=pw) \oplus \gamma(v'=k\cdot inc_0) \oplus \gamma(x=u') \\
&= \textstyle\sum_{u,v,w',x'} \\
&\quad \gamma(u+v=pw+w'+x'-k\cdot inc_0) \oplus \\
&\quad d_0(-inc_0) \oplus e_0(pw+(1-k)\cdot inc_0) \oplus \\
&\quad a_0(-u) \oplus b_0(-v) \oplus c_1(w') \oplus a_2(x') \\
&= \textstyle\sum_{u,v,w,x} \\
&\quad \gamma(u+v=w+x+pw-k\cdot inc_0) \oplus \\
&\quad a_0(-u) \oplus b_0(-v) \oplus d_0(-inc_0) \oplus \\
&\quad a_2(x) \oplus c_1(w) \oplus e_0(pw+(1-k)\cdot inc_0) \\
&= K(\textstyle\sum_{u,v,w,x} \\
&\quad a_0(-u) \oplus b_0(-v) \oplus d_0(-inc_0) \oplus \\
&\quad a_2(x) \oplus c_1(w) \oplus e_0(pw+(1-k)\cdot inc_0))
\end{aligned}
$$

$$
\begin{aligned}
P_1 &= \partial_{\{a_2,b_2,c_1\}}(P_0 \oplus Q_1 \oplus R_2) \\
&= \partial_{\{a_2,b_2,c_1\}}(\\
&\quad \textstyle\sum_{u,v,w,x} \\
&\qquad \gamma(u+v=w+x+pw-k\cdot inc_0) \oplus \\
&\qquad a_0(-u) \oplus b_0(-v) \oplus d_0(-inc_0) \oplus \\
&\qquad a_2(x) \oplus c_1(w) \oplus e_0(pw+(1-k)\cdot inc_0) \oplus \\
&\quad c_1(-pw) \oplus d_1(-inc_1) \oplus \\
&\qquad b_2(k\cdot inc_1) \oplus e_1(pw+(1-k)\cdot inc_1) \oplus \\
&\quad \textstyle\sum_{u,v,w,x} \\
&\qquad \gamma(u+v=w+x) \oplus \\
&\qquad a_2(-u) \oplus b_2(-v) \oplus c_2(w) \oplus a_3(x))
\end{aligned}
$$

$$
\begin{aligned}
&= \textstyle\sum_{u,v,w,x,u',v',w',x'} \\
&\quad \gamma(u' = x) \oplus \gamma(v' = k \cdot inc_1) \oplus \gamma(w = pw) \oplus \\
&\quad \gamma(u + v = w + x + pw - k \cdot inc_0) \oplus \\
&\quad a_0(-u) \oplus b_0(-v) \oplus d_0(-inc_0) \oplus \\
&\qquad e_0(pw + (1-k) \cdot inc_0) \oplus \\
&\quad d_1(-inc_1) \oplus \\
&\qquad e_1(pw + (1-k) \cdot inc_1) \\
&\quad \gamma(u' + v' = w' + x') \oplus \\
&\qquad c_2(w') \oplus a_3(x') \\
&= \textstyle\sum_{u,v,x,w',x'} \\
&\quad \gamma(u + v = x + 2pw - k \cdot inc_0) \oplus \\
&\quad \gamma(x + k \cdot inc_1 = w' + x') \oplus \\
&\quad a_0(-u) \oplus b_0(-v) \oplus d_0(-inc_0) \oplus \\
&\qquad e_0(pw + (1-k) \cdot inc_0) \oplus \\
&\quad d_1(-inc_1) \oplus \\
&\qquad e_1(pw + (1-k) \cdot inc_1) \\
&\quad c_2(w') \oplus a_3(x') \\
&= \textstyle\sum_{u,v,w,x} \\
&\quad \gamma(u + v = w + x + 2pw - k \cdot (inc_0 + inc_1)) \oplus \\
&\quad a_0(-u) \oplus b_0(-v) \oplus \\
&\quad d_0(-inc_0) \oplus e_0(pw + (1-k) \cdot inc_0) \oplus \\
&\quad d_1(-inc_1) \oplus e_1(pw + (1-k) \cdot inc_1) \\
&\quad c_2(w) \oplus a_3(x) \\
&= K(\textstyle\sum_{u,v,w,x} \\
&\quad a_0(-u) \oplus b_0(-v) \oplus \\
&\quad d_0(-inc_0) \oplus e_0(pw + (1-k) \cdot inc_0) \oplus \\
&\quad d_1(-inc_1) \oplus e_1(pw + (1-k) \cdot inc_1) \\
&\quad c_2(w) \oplus a_3(x))
\end{aligned}
$$

B Primer on Tuplix Calculus

This appendix is an excerpt from [6]. For further reading on meadows we refer to [7, 5]. We remark that the operators $+$ for alternative composition and ∂_H for encapsulation stem from the process algebra ACP [2], see also [1, 8]. The summation operator $\sum$ (binding of data variables that generalizes alternative composition) is also part of the specification language μCRL [9], which combines ACP with equationally specified abstract data types.

B.1 Cancellation Meadows

Tuplix Calculus builds on a data type for *quantities*. This data type is required to be a *non-trivial cancellation meadow*, or, equivalently, a *zero-totalized field* [7, 5]. A zero-totalized field is the well-known algebraic structure 'field' with a total operator for division so that the result of division by zero is zero (and, for example, in a 47-totalized field one has chosen 47 to represent the result of all divisions by zero).

A *meadow* is a commutative ring with unit equipped with a total unary operation $(_)^{-1}$ named inverse that satisfies the axioms

$$(x^{-1})^{-1} = x \quad \text{and} \quad x \cdot (x \cdot x^{-1}) = x,$$

and in which $0^{-1} = 0$. For Tuplix Calculus we also require the *cancellation axiom*

$$x \neq 0 \quad \& \quad x \cdot y = x \cdot z \quad \Rightarrow \quad y = z$$

to hold, thus obtaining *cancellation meadows*, which we take as the mathematical structure for quantities, requiring further that $0 \neq 1$ to exclude (trivial) one-point models. These axioms for cancellation meadows characterize exactly the equational theory of zero-totalized fields [5]. The property of cancellation meadows that is exploited in the Tuplix Calculus is that division by zero yields zero, while $x \cdot x^{-1} = 1$ for $x \neq 0$.

We define a *data type* (signature and axioms) for quantities which comprises the constants 0, 1, the binary operators $+$ and $\cdot$, and the unary operators $-$ and $(_)^{-1}$. We often write $x - y$ instead of $x + (-y)$, x/y instead of $x \cdot y^{-1}$, and xy instead of $x \cdot y$, and we shall omit brackets if no confusion can arise following the usual binding conventions. Finally, we use numerals in the common way (2 abbreviates $1 + 1$, etc.). The axiomatization consists of the cancellation axiom

$$x \neq 0 \quad \& \quad x \cdot y = x \cdot z \quad \Rightarrow \quad y = z,$$

the *separation axiom*

$$0 \neq 1,$$

and the following 10 axioms for meadows (see [5]):

$$\begin{aligned}
(x + y) + z &= x + (y + z),\\
x + y &= y + x.\\
x + 0 &= x,\\
x + (-x) &= 0,\\
(x \cdot y) \cdot z &= x \cdot (y \cdot z),\\
x \cdot y &= y \cdot x,\\
1 \cdot x &= x,\\
x \cdot (y + z) &= x \cdot y + x \cdot z,\\
(x^{-1})^{-1} &= x,\\
x \cdot (x \cdot x^{-1}) &= x.
\end{aligned}$$

The following identities are derivable from the axioms for meadows.

$$\begin{aligned}(0)^{-1} &= 0\\ (-x)^{-1} &= -(x^{-1})\\ (x\cdot y)^{-1} &= x^{-1}\cdot y^{-1}\\ 0\cdot x &= 0\\ x\cdot -y &= -(x\cdot y)\\ -(-x) &= x\end{aligned}$$

Furthermore, the cancellation axiom and axiom $x \cdot (x \cdot x^{-1}) = x$ imply the *general inverse law*

$$x \neq 0 \quad \Rightarrow \quad x \cdot x^{-1} = 1$$

of zero-totalized fields.

B.2 Basic Tuplix Calculus

Core Tuplix Calculus (CTC) is parametrized with a nonempty set A of *attributes*. Its signature contains the constants ε (the empty tuplix) and δ (the null tuplix), and two further kinds of atomic tuplices: *entries* (attribute-value pairs) of the form

$$a(t)$$

with $a \in A$, and t a data term, and, for any data term t, the *zero test*

$$\gamma(t)$$

($\gamma \notin A$). Finally, CTC has one binary infix operator: the *conjunctive composition* operator $\oplus$. This operator is commutative and associative. Axioms are in Table 1.

In CTC, a tuplix is a conjunctive composition of tests and entries, with ε representing an empty tuplix, and δ representing an erroneous situation which nullifies the entire composition. Entries with the same attribute can be combined to a single entry containing the sum of the quantities involved.

A zero test $\gamma(t)$ acts as a conditional: if the argument t equals zero, then the test is void and disappears from conjunctive compositions. If the argument is not equal to zero, the test nullifies any conjunctive composition containing it. Observe how we exploit the property of zero-totalized fields that t/t is always defined, and that the division t/t yields zero if t equals zero, and 1 otherwise. Further note that an equality test $t = s$ can be expressed as $\gamma(t - s)$.

A tuplix term is *closed* if it is does not contain tuplix variables and also does not contain data variables. A tuplix term is *tuplix-closed* if it does not contain tuplix variables (but it may contain data variables).

The tuplix calculus is two-sorted. On the tuplix side we have the axioms T1–T10 and we use the proof rules of equational logic. On the data side, we

Table 1: Axioms for Basic Tuplix Calculus

$X \oplus Y = Y \oplus X$	(T1)
$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$	(T2)
$X \oplus \varepsilon = X$	(T3)
$X \oplus \delta = \delta$	(T4)
$a(x) \oplus a(y) = a(x + y)$	(T5)
$\gamma(x) = \gamma(x/x)$	(T6)
$\gamma(0) = \varepsilon$	(T7)
$\gamma(1) = \delta$	(T8)
$\gamma(x) \oplus \gamma(y) = \gamma(x/x + y/y)$	(T9)
$\gamma(x - y) \oplus a(x) = \gamma(x - y) \oplus a(y)$	(T10)
$X + Y = Y + X$	(C1)
$(X + Y) + Z = X + (Y + Z)$	(C2)
$X + X = X$	(C3)
$X + \delta = X$	(C4)
$X \oplus (Y + Z) = (X \oplus Y) + (X \oplus Z)$	(C5)
$\gamma(x) + \gamma(y) = \gamma(xy)$	(C6)

refrain from giving a precise proof theory. The rule DE lifts valid data identities to the tuplix calculus: for all (open) data terms t and s,

$$\mathcal{D} \models t = s \quad \text{implies} \quad \gamma(t) = \gamma(s), \tag{DE}$$

where $\mathcal{D}$ (a non-trivial cancellation meadow) is our model of the data type. This axiom system with axioms T1–T10 plus proof rule DE is denoted by CTC.

The axiom system CTC is extended to Basic Tuplix Calculus (BTC), by addition of the binary operator + called *alternative composition* or *choice* to the signature, and by adoption of axioms C1–C6 (see Table 1).

The following two proof rules are derivable:

$$\mathcal{D} \models t = s \quad \text{implies} \quad P[t/x] = P[s/x],$$

and

$$P \between \gamma(x - t) = P[t/x] \between \gamma(x - t),$$

for tuplix terms P and with substitution $P[t/x]$ defined as usual for two-sorted equational logic (replacement of all data variables x in P by t).

B.3 Zero-Test Logic

We present some observations on the use of the zero-test operator which lead to a simple logic.

First, the empty tuplix ε with $\varepsilon = \gamma(0)$ by axiom T7 may be read as 'true', and the null tuplix δ with $\delta = \gamma(1)$ by axiom T8 may be read as 'false'.

Negation. Define the test 'not $x = 0$' by

$$\widetilde{\gamma}(x) \stackrel{\text{def}}{=} \gamma(1 - x/x).$$

Conjunctive composition of tests may be read as logical conjunction:

$$\gamma(x) \between \gamma(y) \stackrel{(\text{T9})}{=} \gamma(x/x + y/y)$$

tests '$x = 0$ and $y = 0$'.

Alternative composition of tests may be read as logical disjunction:

$$\gamma(x) + \gamma(y) \stackrel{(\text{C6})}{=} \gamma(x \cdot y)$$

tests '$x = 0$ or $y = 0$'.

A *formula* would then be a tuplix-closed (no tuplix variables) BTC term without entries. Any formula can be expressed as a single test $\gamma(t)$ using axioms T7–T9 and C6, and the definition of negation. We find that this logic has all the usual properties. Clearly, conjunction and disjunction are commutative, associative, and idempotent, and it is not difficult to derive distributivity, absorption, and double negation elimination. As usual, implication can be defined in terms of negation and disjunction:

$$\widetilde{\gamma}(x) + \gamma(y) = \gamma((1 - x/x) \cdot y)$$

tests '$x = 0$ implies $y = 0$'.

B.4 Generalized Alternative Composition and Auxiliary Operators

The *generalized alternative composition (*or: *summation) operator* $\sum_x$ is a unary operator that *binds* data variable x and can be seen as a data-parametric generalization of the alternative composition operator $+$. We add this binder to the signature of BTC and write $FV(P)$ for the set of free data variables occurring in tuplix term P. We write $Var(t)$ for the set of data variables occurring in data term t (there is no variable binding within data terms). Define substitution $P[t/x]$ as: replace every free occurrence of data variable x in tuplix term P by the data term t, such that no variables of t become bound in these replacements. E.g., recall the proof rule

$$P \oplus \gamma(x - t) = P[t/x] \oplus \gamma(x - t).$$

This rule remains sound in the setting with summation, but application of the rule may require the renaming of bound variables in P, so that the substitution can be performed. When considering substitutions we implicitly assume that bound variables are renamed properly. The axiom schemes for summation are listed in Table 2.

Auxiliary Operators. For BTC with summation, we define three auxiliary operators: scalar multiplication, clearing, and encapsulation. Axioms are listed in Table 2.

- Scalar multiplication $t \cdot P$ multiplies the quantities contained in entries in tuplix term P by t. Axiom Sc7 is an axiom scheme with t ranging over data terms and P ranging over tuplix terms.
- Clearing: For set of attributes $I \subseteq A$, the operator $\varepsilon_I(X)$ renames all entries of X with attribute in I to ε. It "clears" the attributes contained in I. For a set of attributes $J \subseteq A$ we further define

  $$Select_J(X) \stackrel{\text{def}}{=} \varepsilon_{A \setminus J}(X).$$

  This function allows to focus on those entries with attribute from J.
- Encapsulation can be seen as 'conditional clearing'. For set of attributes $H \subseteq A$, the operator $\partial_H(X)$ encapsulates all entries in X with attribute $a \in H$. That is, for $a \in H$, if the accumulation of quantities in entries with attribute a equals zero, the encapsulation on a is considered successful and the a-entries are *cleared* (become ε); if the accumulation is not equal to zero, they become null (δ). This accumulation of quantities is computed per alternative: the encapsulation operator distributes over alternative composition.

  We further define

  $$\partial_{H \cup H'}(X) \stackrel{\text{def}}{=} \partial_H \circ \partial_{H'}(X).$$

Table 2: Axiom schemes for generalization and auxiliary operators. Terms P and Q range over tuplix terms and t ranges over data terms.

Axiom	Condition	Label
$\sum_x P = P$	if $x \notin FV(P)$	(S1)
$\sum_x P = \sum_y P[y/x]$	if $y \notin FV(P)$	(S2)
$\sum_x (P \oplus Q) = P \oplus \sum_x Q$	if $x \notin FV(P)$	(S3)
$\sum_x (P + Q) = \sum_x P + \sum_x Q$		(S4)
$\sum_x \gamma(x - t) = \varepsilon$	if $x \notin Var(t)$	(S5)
$\sum_x \widetilde{\gamma}(x - t) = \varepsilon$	if $x \notin Var(t)$	(S6)
$x \cdot \varepsilon = \varepsilon$		(Sc1)
$x \cdot \delta = \delta$		(Sc2)
$x \cdot \gamma(y) = \gamma(y)$		(Sc3)
$x \cdot a(y) = a(x \cdot y)$		(Sc4)
$x \cdot (X \oplus Y) = x \cdot X \oplus x \cdot Y$		(Sc5)
$x \cdot (X + Y) = x \cdot X + x \cdot Y$		(Sc6)
$t \cdot \sum_y P = \sum_y (t \cdot P)$	if $y \notin Var(t)$	(Sc7)
$\varepsilon_I(\varepsilon) = \varepsilon$		(Cl1)
$\varepsilon_I(\delta) = \delta$		(Cl2)
$\varepsilon_I(\gamma(x)) = \gamma(x)$		(Cl3)
$\varepsilon_I(a(x)) = \begin{cases} \varepsilon & \text{if } a \in I \\ a(x) & \text{otherwise} \end{cases}$		(Cl4)
$\varepsilon_I(X \oplus Y) = \varepsilon_I(X) \oplus \varepsilon_I(Y)$		(Cl5)
$\varepsilon_I(X + Y) = \varepsilon_I(X) + \varepsilon_I(Y)$		(Cl6)
$\varepsilon_I(\sum_x P) = \sum_x (\varepsilon_I(P))$		(Cl7)
$\partial_H(\varepsilon) = \varepsilon$		(E1)
$\partial_H(\delta) = \delta$		(E2)
$\partial_H(\gamma(x)) = \gamma(x)$		(E3)
$\partial_H(a(x)) = \begin{cases} \gamma(x) & \text{if } a \in H \\ a(x) & \text{if } a \notin H \end{cases}$		(E4)
$\partial_H(X \oplus \partial_H(Y)) = \partial_H(X) \oplus \partial_H(Y)$		(E5)
$\partial_H(X + Y) = \partial_H(X) + \partial_H(Y)$		(E6)
$\partial_H(\sum_x P) = \sum_x (\partial_H(P))$		(E7)

References

[1] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[2] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control* 60(1–3):109–137, 1984.

[3] J.A. Bergstra, S. Nolst Trenité and M.B. van der Zwaag. Towards a formalization of budgets. arXiv.org, arXiv:0802.3617v1 [cs.LO], 2008.

[4] J.A. Bergstra, S. Nolst Trenité and M.B. van der Zwaag. UvA budget allocation model. Report PRG0805, Section Software Engineering, University of Amsterdam, 2008.

[5] J.A. Bergstra and A. Ponse. A generic basis theorem for cancellation meadows. arXiv.org, arXiv:0803.3969v2 [math.RA], 2008.

[6] J.A. Bergstra, A. Ponse and M.B. van der Zwaag. Tuplix Calculus. arXiv.org, arXiv:0712.3423v1 [cs.LO], 2007.

[7] J.A. Bergstra and J.V. Tucker. The rational numbers as an abstract data type. *Journal of the ACM* 54(2), 2007.

[8] W. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science, Springer-Verlag, 2000.

[9] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In: A. Ponse, C. Verhoef and S.F.M. van Vlijmen (editors), *Algebra of Communicating Processes '94*, pages 26–62, Workshops in Computing Series, Springer-Verlag, 1995.

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0806] J.A. Bergstra and C.A. Middelburg, *Data Linkage Algebra, Data Linkage Dynamics, and Priority Rewriting,* Programming Research Group - University of Amsterdam, 2008.

[PRG0805] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *UvA Budget Allocatie Model,* Programming Research Group - University of Amsterdam, 2008.

[PRG0804] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Sequential Poly-Threading,* Programming Research Group - University of Amsterdam, 2008.

[PRG0803] J.A. Bergstra and C.A. Middelburg, *Thread Extraction for Polyadic Instruction Sequences,* Programming Research Group - University of Amsterdam, 2008.

[PRG0802] A. Barros and T. Hou, *A Constructive Version of AIP Revisited,* Programming Research Group - University of Amsterdam, 2008.

[PRG0801] J.A. Bergstra and C.A. Middelburg, *Programming an Interpreter Using Molecular Dynamics,* Programming Research Group - University of Amsterdam, 2008.

[PRG0713] J.A. Bergstra, A. Ponse, and M.B. van der Zwaag, *Tuplix Calculus,* Programming Research Group - University of Amsterdam, 2007.

[PRG0712] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *Towards a Formalization of Budgets,* Programming Research Group - University of Amsterdam, 2007.

[PRG0711] J.A. Bergstra and C.A. Middelburg, *Program Algebra with a Jump-Shift Instruction,* Programming Research Group - University of Amsterdam, 2007.

[PRG0710] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences with Dynamically Instantiated Instructions,* Programming Research Group - University of Amsterdam, 2007.

[PRG0709] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences with Indirect Jumps,* Programming Research Group - University of Amsterdam, 2007.

[PRG0708] B. Diertens, *Software (Re-)Engineering with PSF III: an IDE for PSF,* Programming Research Group - University of Amsterdam, 2007.

[PRG0707] J.A. Bergstra and C.A. Middelburg, *An Interface Group for Process Components,* Programming Research Group - University of Amsterdam, 2007.

[PRG0706] J.A. Bergstra, Y. Hirschfeld, and J.V. Tucker, *Skew Meadows,* Programming Research Group - University of Amsterdam, 2007.

[PRG0705] J.A. Bergstra, Y. Hirschfeld, and J.V. Tucker, *Meadows,* Programming Research Group - University of Amsterdam, 2007.

[PRG0704] J.A. Bergstra and C.A. Middelburg, *Machine Structure Oriented Control Code Logic (Extended Version),* Programming Research Group - University of Amsterdam, 2007.

[PRG0703] J.A. Bergstra and C.A. Middelburg, *On the Operating Unit Size of Load/Store Architectures,* Programming Research Group - University of Amsterdam, 2007.

[PRG0702] J.A. Bergstra and A. Ponse, *Interface Groups and Financial Transfer Architectures,* Programming Research Group - University of Amsterdam, 2007.

[PRG0701] J.A. Bergstra, I. Bethke, and M. Burgess, *A Process Algebra Based Framework for Promise Theory,* Programming Research Group - University of Amsterdam, 2007.

[PRG0610] J.A. Bergstra and C.A. Middelburg, *Parallel Processes with Implicit Computational Capital,* Programming Research Group - University of Amsterdam, 2006.

[PRG0609] B. Diertens, *Software (Re-)Engineering with PSF II: from architecture to implementation,* Programming Research Group - University of Amsterdam, 2006.

[PRG0608] A. Ponse and M.B. van der Zwaag, *Risk Assessment for One-Counter Threads,* Programming Research Group - University of Amsterdam, 2006.

[PRG0607] J.A. Bergstra and C.A. Middelburg, *Synchronous Cooperation for Explicit Multi-Threading,* Programming Research Group - University of Amsterdam, 2006.

[PRG0606] J.A. Bergstra and M. Burgess, *Local and Global Trust Based on the Concept of Promises,* Programming Research Group - University of Amsterdam, 2006.

[PRG0605] J.A. Bergstra and J.V. Tucker, *Division Safe Calculation in Totalised Fields,* Programming Research Group - University of Amsterdam, 2006.

[PRG0604] J.A. Bergstra and A. Ponse, *Projection Semantics for Rigid Loops,* Programming Research Group - University of Amsterdam, 2006.

[PRG0603] J.A. Bergstra and I. Bethke, *Predictable and Reliable Program Code: Virtual Machine-based Projection Semantics (submitted for inclusion in the Handbook of Network and Systems Administration),* Programming Research Group - University of Amsterdam, 2006.

[PRG0602] J.A. Bergstra and A. Ponse, *Program Algebra with Repeat Instruction,* Programming Research Group - University of Amsterdam, 2006.

[PRG0601] J.A. Bergstra and A. Ponse, *Interface Groups for Analytic Execution Architectures,* Programming Research Group - University of Amsterdam, 2006.

[PRG0505] B. Diertens, *Software (Re-)Engineering with PSF,* Programming Research Group - University of Amsterdam, 2005.

[PRG0504] P.H. Rodenburg, *Piecewise Initial Algebra Semantics,* Programming Research Group - University of Amsterdam, 2005.

[PRG0503] T.D. Vu, *Metric Denotational Semantics for BPPA,* Programming Research Group - University of Amsterdam, 2005.

[PRG0502] J.A. Bergstra, I. Bethke, and A. Ponse, *Decision Problems for Pushdown Threads,* Programming Research Group - University of Amsterdam, 2005.

[PRG0501] J.A. Bergstra and A. Ponse, *A Bypass of Cohen's Impossibility Result,* Programming Research Group - University of Amsterdam, 2005.

[PRG0405] J.A. Bergstra and I. Bethke, *An Upper Bound for the Equational Specification of Finite State Services,* Programming Research Group - University of Amsterdam, 2004.

[PRG0404] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Strategic Interleaving,* Programming Research Group - University of Amsterdam, 2004.

[PRG0403] B. Diertens, *A Compiler-projection from PGLEc.MSPio to Parrot,* Programming Research Group - University of Amsterdam, 2004.

[PRG0402] J.A. Bergstra and I. Bethke, *Linear Projective Program Syntax,* Programming Research Group - University of Amsterdam, 2004.

[PRG0401] B. Diertens, *Molecular Scripting Primitives,* Programming Research Group - University of Amsterdam, 2004.

[PRG0302] B. Diertens, *A Toolset for PGA,* Programming Research Group - University of Amsterdam, 2003.

[PRG0301] J.A. Bergstra and P. Walters, *Projection Semantics for Multi-File Programs,* Programming Research Group - University of Amsterdam, 2003.

[PRG0201] I. Bethke and P. Walters, *Molecule-oriented Java Programs for Cyclic Sequences,* Programming Research Group - University of Amsterdam, 2002.

The above reports and more are available through the website: www.science.uva.nl/research/prog/

Electronic Report Series

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
the Netherlands

www.science.uva.nl/research/prog/